Non-Linear Programming Research Project
Optimisation Theory and Applications

Felix Newport-Mangell
fn17351@bristol.ac.uk

Gradient descent for univariate Linear Regression

A fundamental technique in machine learning is the prediction of variables with linear relationships. The most basic form of this prediction is for univariate functions, where predictions can be made based on a single parameter and a bias term.

These linear relationships are of the form

$$h(\boldsymbol{x}) = \theta_0 * x_0 + \theta_1 * x_1 \quad (1)$$

or

$$h(\boldsymbol{x}) = \boldsymbol{\theta} \cdot \boldsymbol{x} \quad (2)$$

Where $h(\boldsymbol{x})$ is a numerical prediction (hypothesis) made by the linear model characterised by the input feature values $\boldsymbol{x}$ ($= [x_0 \ x_1]$) and 'weights' of the model $\boldsymbol{\theta}$ ($= [\boldsymbol{\theta_0} \ \boldsymbol{\theta_1}]$).

The example used in this research project is a dummy dataset created using the sci-kit learn [1] module 'make_regression' that can be used to train a linear model. The dependent and independent variable (feature and target respectively) in this case were arbitrarily chosen and simulate a relationship between housing prices (£'000s) and the property's floorspace in square metres.

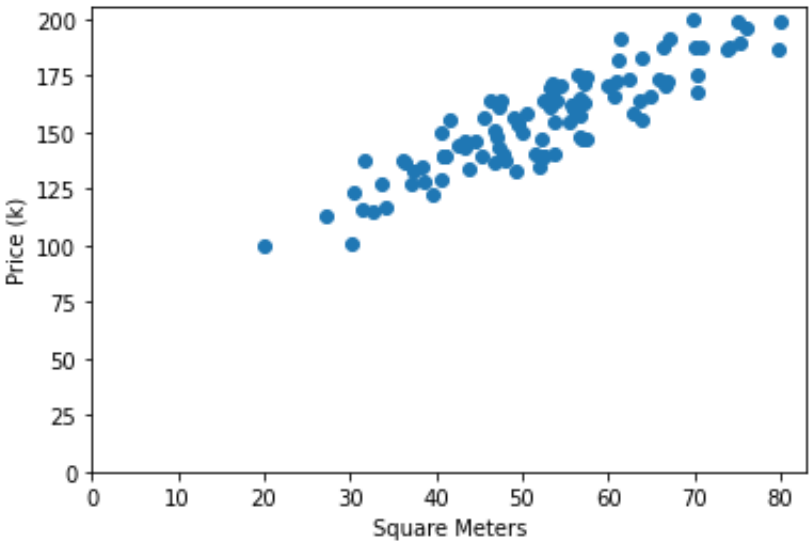


Figure 1 Plot of the generated data

The challenge in making accurate predictions is in choosing the right parameters $\boldsymbol{\theta}$. Using a technique known as Linear Regression [2], these parameters can be learned algorithmically.

This research project's focus is to implement the Conjugate Gradient Descent (CGD) algorithm developed in part A of the assignment in a Linear Regression. For more information on how Linear Regression works, please consult Ref.2. The rest of the report will assume working understanding of the concepts used in Linear Regression.

Function to minimise

The function to minimise using CGD is the cost function of predictions, represented by the mean squared error over all examples $x_1, x_2, \dots, x_n$ defined as:

$$J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}((h(x_i) - y_i)^2 \tag{3}$$

$J(\theta_0, \theta_1) = Cost\ function$

$m = number\ of\ examples$

$h(x_i) = prediction\ of\ example\ i$

$y_i = true\ value\ of\ example\ i$

This is a non-linear function that can be optimised using conjugate gradient descent.

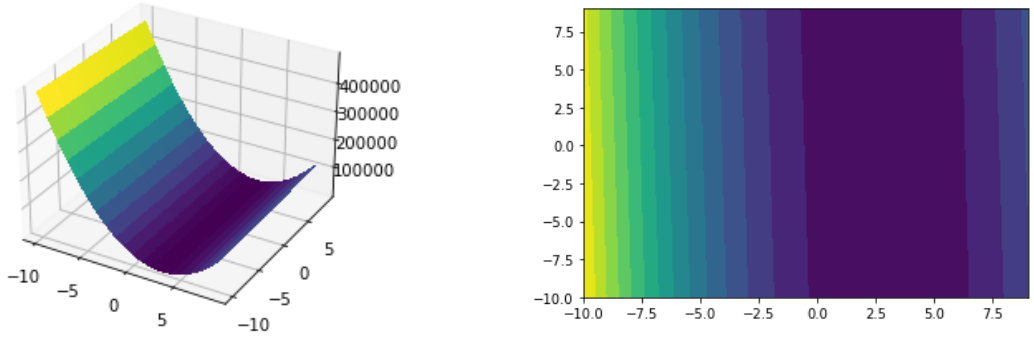


Figure 2 Surface and Contour plots of the cost function for varying values of θ_0 and θ_1

Optimising with Conjugate Gradient Descent

This function was input into the CGD algorithm developed in part A of the assignment with the following parameters [3]:

Starting position: $\boldsymbol{\theta} = [0, 1]$

eps = 0.01

eta = 0.1

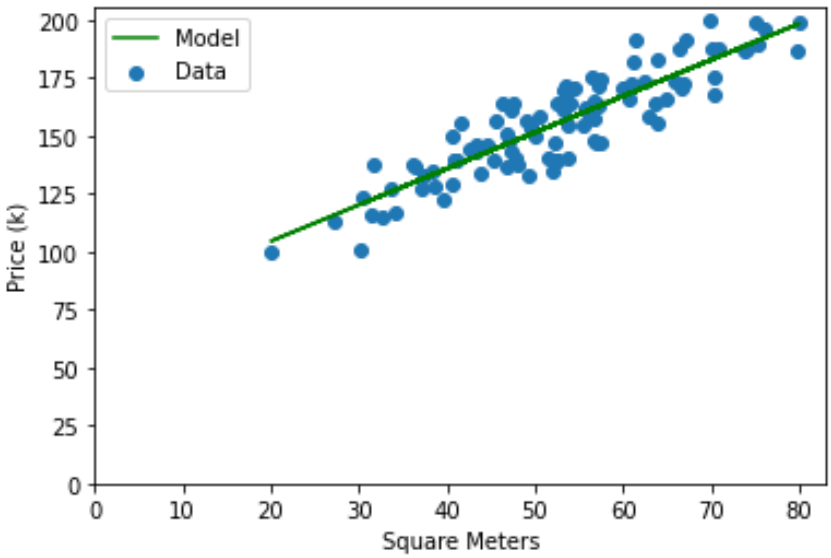
Line search max iterations (i_max) = 100

CGD max iterations (j_max) = 1000

At termination, the theta found to minimise the cost function was:

$$\boldsymbol{\theta} = [73.2 \;\; 1.56]$$

Predictions made by the model with these weights can be visualised and super-imposed on the data plot. The function is optimal with a MSE value of 95.38 having started with a value of 10,699 with the initialised theta.



The accuracy of this linear model can be compared with a state-of-the-art Linear Regression model using sci-kit learn's implementation [1]. This returns a MSE value of 95.36 with optimal parameters theta.

Comparing these results confirms this CGD implementation's capability to optimise (minimise) convex non-linear functions.

References

[1] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F.. Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B. 'Scikit-learn: Machine Learning in Python', Journal of Machine Learning Research 12 (2011) pages 2825-2830.

[2] Brownlee, J. (2016) 'Linear Regression for Machine Learning', Machine Learning Mastery, March 25th, Available at: https://machinelearningmastery.com/linear-regression-for-machine-learning/. Accessed 08/12/2021

[3] Newport-Mangell, F. (2021) 'readme.txt', Optimization theory and applications coursework assignment 2: non-linear programming submission. December 8th.

Appendix

CGD Algorithm iteration results

```
Alpha at end of line search: 1.6962920848248801e-06
AFTER CG ALGORITHM ITERATION 1: THETA = [0.03484948 2.86042285]

Alpha at end of line search: 1.6962920848248801e-06
AFTER CG ALGORITHM ITERATION 2: THETA = [0.03655153 2.88117515]

Alpha at end of line search: 0.0003901434166249718
AFTER CG ALGORITHM ITERATION 3: THETA = [0.34351094 2.8958733 ]

Alpha at end of line search: 0.00040421510059749234
AFTER CG ALGORITHM ITERATION 4: THETA = [73.05537816  1.59561554]

Alpha at end of line search: 1.6962920848248801e-06
AFTER CG ALGORITHM ITERATION 5: THETA = [73.72789309  1.55381502]

Alpha at end of line search: 1.6962920848248801e-06
AFTER CG ALGORITHM ITERATION 6: THETA = [73.7279983   1.55338793]

Alpha at end of line search: 0.026320879868528158
AFTER CG ALGORITHM ITERATION 7: THETA = [73.4075648  1.5613871]

Alpha at end of line search: 6.1018873206235235e-06
AFTER CG ALGORITHM ITERATION 8: THETA = [72.59041356  1.57387184]

Alpha at end of line search: 1.739993388216691e-06
AFTER CG ALGORITHM ITERATION 9: THETA = [72.59040375  1.57388674]
```

Code

```python
# generate data
from sklearn.datasets import make_regression

np.random.seed(0)
X, y = make_regression(n_samples=100, n_features=1, noise=20)
X = np.interp(X, (X.min(), X.max()), (20, 80))
y = np.interp(y, (y.min(), y.max()), (100, 200))
X = np.hstack((np.ones(shape=(len(X), 1)), X))
```

```python
print('X:\n', X[0:5])
print('\ny:\n', y[0:5])
```

```
X:
 [[ 1.         47.28865294]
 [ 1.         63.91228558]
 [ 1.         56.76742354]
 [ 1.         41.6454787 ]
 [ 1.         40.71688545]]

y:
 [143.52729978 155.38820723 157.57070549 155.61543981 139.3689206 ]
```

```python
# make initial guess for parameters of univariate linear model (bias: theta_0, weight: theta_1)
theta_0 = 0
theta_1 = 1
W = [theta_0, theta_1]

# define hypotheses
h = lambda W : W[0] * X[:,0] + W[1] * X[:,1]
err_func = lambda W : h(W)-y
J_func = lambda W : 1/len(X) * np.inner(err_func(W), err_func(W))

# calculate cost function
err = h(W)-y
J = np.inner(err, err)
```